COMP90015 Project 2 Report

Yang Song(yasong2) 1197417

Xiaoyu Cheng(xiaoycheng) 1147378

Group contribution

Xiaoyu was responsible for transforming the existing chat program into a decentralized chat application and writing the second part and the third part of the report. Yang was responsible for programming the extended feature and writing the first part of the report. We managed to maintain good communication throughout the project.

1. Extended Feature

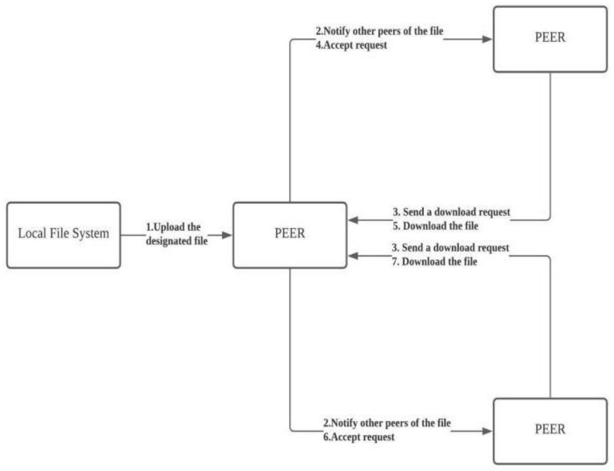
Local File System
1.Upload the designated file
PEER
2.Notify other peers of the file
4.Accept request
PEER
3. Send a download request
5. Download the file
3. Send a download request
7. Download the file
2.Notify other peers of the file
6.Accept request
PEER

Figure 1. FSM diagram explaining P2P file attachment transmission

The chosen extended feature for our project is File Attachment. This feature would allow the peers to upload files from the local file system and also download files from other peers connected to the P2P network. The upload and download process will be using buffered input streams and buffered output streams. Most file types would be accepted, including mp3, pdf, and txt, as long as the 'isfile()' function, which determines whether the abstract filename denotes a file or Directory, returns true. The user's file attachment would be sent from the user's computer to to a peer connected to the P2P network. The peer would only alert other peers in the same room about the newly uploaded file attachment, similar to a message. After receiving this message, the other peers in the room could send a request for downloading the file attachment according to the messages sent to them. Since the download process is peer-to-peer, only one file transmission process is allowed between two peers at a time. However, there might be multiple peers requesting the same file attachment, but there could only be one uploader uploading the file attachment. In this case, when the uploader peer is occupied, the download request from other peers would not be accepted. Therefore the downloader peer has to wait until the uploader peer is available for file transmission if there are multiple download requests.

Several major challenges of distributed systems were revealed when implementing this feature. The first is failure handling. Failure handling prevents programs from producing unwanted incorrect results which may lead to irreparable damage when faults occur in hardware or software. For example, when the peer which uploaded the file attachment gets disconnected from the network, at this moment if there are other peers still downloading the file, the download process would be terminated unexpectedly with force, which could lead those peers to crash. To handle this kind of failure, our system should terminate the download thread automatically, catch the exception, and also notify the downloader(s) about the reason for download termination. Another situation is when the downloader disconnects without notification, the download process should also be terminated silently by the system and prevent the uploader peer from crashing.

The next challenge for file attachment is transparency. Transparency is defined as the concealment from the user of the separation of components in a distributed system, therefore hiding the complexity of the system. In our system, all the data transmission process of the extended feature is hidden from the user. They are only provided with two simple commands: #upload and #download. Upload command will allow the user to input the file path and upload the designated file, and the download command will trigger the program to start downloading the files from other connected peers and store it at the path entered by the user. Forbidding users from accessing complicated backend codes will significantly reduce the risk of misoperation or misinterpretation.

We also recognize potential concurrency problems during file transmission. In our system, there is a possibility that several peers will attempt to access a shared resource at the same time. Since we are using a peer-to-peer structure instead of a client/server structure, it is impossible for multiple peers to download a file from a single peer at the same time. Hence, the peers will take turns to download the file. In this case, the peer's code should work like this: all file transmissions are being processed using a uploader's thread and a downloader's thread. The uploader's thread will be working on uploading the file from local to the connected peer. The uploader peer will only accept new download requests if its thread is currently free. The downloader can send a request to detect whether the uploading peer has a free thread, and the downloader also has its thread waiting to download the file.

Another issue is the potential thread collision when uploading a file. To prevent this problem, it is necessary to use a synchronized method for upload. When processing the request for downloading the file, the peer which responds to the request would initiate a thread for each request to handle file transmission, and the thread must pass the synchronized method, therefore at the same time there can be only one thread running at a time. Any other threads must wait for this thread to finish executing the synchronized method, then they are allowed to run the synchronized method. This ensures the security of each thread.

Finally, since we have to access the local file system on the user's device, data security is also one of our concerns. The first measure is always closing the resource after use. Normally, when running a Java application, it would invoke different types of resources such as files, streams and sockets. If we don't clean up the resources we open, it can create some potential vulnerability. For instance, if a stream is left unclosed, there might be a chance for a resource leak. In addition, the operating system might recognize that the resources used by the application are no longer needed, therefore locking the files/etc and causing trouble for the user. For the sake of safety and user experience, we should always handle these resources properly and free them if we don't need them anymore.

Of course, there are still some improvements that can be done on the security part. For example, we can limit the program to read-only mode while uploading. This could also help secure data integrity.

2. Decentralized chat application

In the decentralized chat application, each peer needs to serve as a server and a client. The peer runs two concurrent threads after startup, one to accept user input and the other to wait for TCP connection requests. When accepting an incoming TCP connection or initiating an outgoing connection, the peer runs a separate thread to handle connection with another peer. When the peer disconnects, its incoming and outgoing connections are closed on both sides.

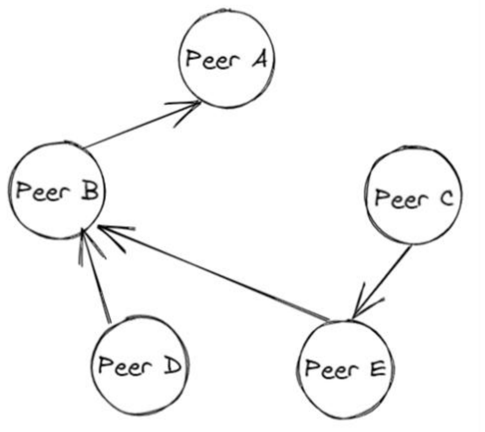

Figure 2. Partial mesh network topology for P2P chat application

As the peers can issue connection or disconnection at any time, the network topology may change continuously. But our decentralized chat application follows a partial mesh pattern, with some peers having multiple connected peers and others having 0 or 1 connected peer. (Figure 2) As per the connection constraint, a peer can only have one outgoing TCP connection with another peer at any given time. As a result, in the network graph, a peer has an out-degree between 0 and 1 for outgoing connection, and an in-degree of between 0 and n for incoming connections, where n ≤number of available ports on the peer.

The communication pattern is peer-to-peer. Assume a P2P connection is established. When issuing remote commands including #list, #listneighbors and #who, the message complexity is O(1) because the peer only sends the command to its server which responds back directly. When issuing #delete/ join/ quit commands, the message complexity becomes O(number of peers) because the server needs to broadcast room change messages to peers in the relevant room. For file sharing, files are stored at the provider peer and the provider can notify all peers in the same room created by itself, with O(number of peers) during broadcast and O(1) when transferring the file on a one-to-one basis.

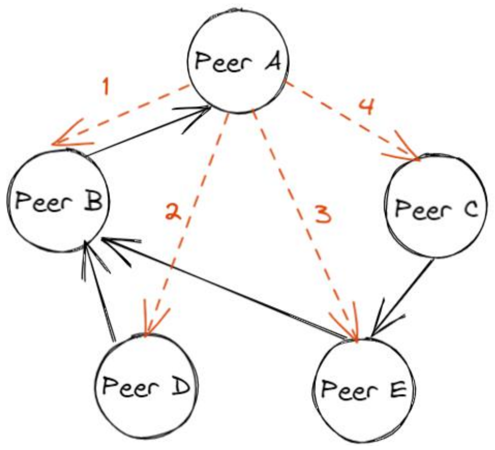

Figure 3. Partial mesh network topology when issuing #searchnetwork command

When issuing #searchnetwork command, the peer iteratively initiates transient connections with accessible peers in the network to find other peers and then disconnect. The network topology remains partial mesh and the out-degree is 1 for each transient connection. (Figure 3) As for the communication pattern, the peer connects to one available peer at a time and issues #list and #listneighbors commands, therefore the overall message complexity is O(number of accessible peers). In our unstructured P2P application, each peer only keeps a dynamic list of ip addresses and listening ports of connected peers but does not have a global list of all peers. Although the search network feature allows exploring other peers and encourages a denser network connectivity and exchange of resources (i.e., file sharing), a peer may not be able to retrieve all peers in the network and the number of accessible peers is limited by potential network partition.

3. Failure model

When compared to a centralized client-server system, the decentralized system is more robust to failure since it eliminates the single point of failure vulnerability. However, as our chat service uses an unstructured P2P architecture, individual peer failures are unrecoverable which may pose a significant impact on system performance, such as resource accessibility. (Samant et al., 2004)

As discussed in the lecture, assumedly the mean time to failure (MTTF) for each independent process is 1 week and given by the equation (failure rate= 1/MTTF), the failure rate is 0.14 process failure per day. The failure rate increases linearly with the increasing number of peers joining the network. Suppose our decentralized chat application has 1000 active peers and thus 1000 processes, it will experience 140 process failures per day on average, or 5.8 failures per hour. It suggests that this P2P system becomes particularly prone to failure when the number of connecting peers grows large.

In our P2P implementation, each individual peer may have a set of connections with other peers and keeps relevant peer address information, as well as a set of resources to offer (such as files). Each resource is stored and delivered by a single provider with zero redundancy.

In the event of process failure, i.e. when a user terminates the peer which hosts several clients, the clients are forced to exit their current room and if there are any message passing in progress, either in the room or with the server, the clients may be unable to receive the packets. Similarly, if the server was sending a file to a client, file transfer is halted, and other pending download requests are ignored. As a result, the file is no longer accessible to the rest of the peers. Process failure may also lead to network partition. For example, if a peer with numerous peer connections is shut down, some clients which previously only had 1 connection to the server will no longer be able to access and connect to other peers by issuing the commands (#listneighbors or #searchnetwork) to the server. As a result, these clients become isolated network components.

In the event of network failure, packets may be dropped largely due to full queueing under high load. However, since our P2P chat system uses the Transmission Control Protocol (TCP) to provide reliable, ordered, and error-checked delivery with congestion control for both message relay and file transfer, this problem may be mitigated.

Citation:

1. Bouchrika, I. (2021). Challenges for a Distributed System. Retrieved 31 October 2021, from https://www.ejbtutorial.com/distributed-systems/challenges-for-a-distributed-system

2. Topolnik, M. (2021). Is closing the resources always important?. Retrieved 31 October 2021, from https://stackoverflow.com/questions/18002896/is-closing-the-resources-always-important

3. Samant, K., & Bhattacharyya, S. (2004). Topology, search, and fault tolerance in unstructured P2P networks. *37th Annual Hawaii International Conference on System Sciences, 2004.*